University of Alberta

An Emulated IO Subsystem for Reactive Applications

by

Elvira Akhmetshina

A thesis submitted to the Faculty of Graduate Studies and research in partial fulfillment of the requirements for the degree of Master of Science

Department of Computing Science

Edmonton, Alberta

Fall 2003

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisisitons et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-87707-8
Our file Notre référence
ISBN: 0-612-87707-8

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

Canadä

University of Alberta

Library Release Form

Name of Author: Elvira Akhmetshina

Title of Thesis: An Emulated IO Subsystem For Reactive Applications

Degree: Master of Science

Year this Degree Granted: 2003

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Date: 2003/10/1

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **An Emulated IO Subsystem for Reactive Applications** submitted by **Elvira Akhmetshina** in partial fulfillment of the requirements of **Master of Science**.

—
ski

—
ms

—
Lu

—
nch

Date: _____

Abstract

Reactive systems perceive and respond to changes in their physical environment via peripheral IO devices. These physical devices are modeled by software specification systems using simulation and emulation techniques. SMURPH, a System for Modeling Unslotted Real-time PHenomena, laid a groundwork in this direction by developing a methodology for representing the physical environment as an emulated event-based system that serves to describe networking devices at the Medium Access Control level. The approach was adopted in PicOS, a multitasking embedded operating system founded on a principle of co-operative IO-intensive processes. In extension of the SMURPH paradigm, we present a method for deploying a reactive event-driven platform in a virtual embedded environment. By emulating IO at the lowest software level (IO registers) we achieve a realistic and yet completely virtual system of peripheral devices controlled by the PicOS kernel.

The IO system developed in this thesis is part of a larger project that aims at constructing a platform for reactive systems, primarily wireless embedded applications. In the emerging era of pervasive computing, when inexpensive ordinary devices and appliances communicate wirelessly at a new, never seen before, ubiquitous level, study of wireless network platforms has attained a research interest of growing importance.

This work, like any other, is dedicated to

the Creator, the Redeemer, and the Counselor

Acknowledgements

First of all, I would like to acknowledge the love and support of my Heavenly Father in my work in this program, and without Who nothing ever would be possible. I am very grateful to Professor Pawel Gburzynski for his endless expertise, support, and guidance. Many thanks go to Professor Janelle Harms for her kind assistance in registering for and very helpful teaching of the first operating systems course. She and other members of the Committee, Professors Paul Lu and Adam Lynch, provided helpful and much appreciated comments. I am indebted to Frederick Vizeacoumar for his encouragement and support during the hectic days before the defense.

Table of Contents

1. Introduction 1

2. Reactive Methodology 7

A Paradigm for Modeling Reactive Physical Systems 7

A Control Specification for Distributed Devices 12

Contrasts Between Preemptive Systems and PicOS 14

Synchronization in Preemptive Systems 16

Kernel Services for Reactive Applications 18

Using PicOS as a Platform for Embedded Applications 21

3. The Virtual Machine 29

Introduction 29

Constituents 30

MMU and Memory Devices 32

CPU's Peripheral IO Ports 37

Clock Devices 40

Ethernet 45

DUART 47

Radios 49

LCD 52

LEDs 52

Beeper 53

4. Process Scheduling Using the Virtual Machine 54

5. Conclusions and Future Work 61

Bibliography 63

List of Figures

1.1. The Place of Emulated IO in the Platform Architecture 4

2.1. The Process Model in SMURPH 8

2.2. The Perform Method in SMURPH 9

2.3. The User-Visible Class Hierarchy in SMURPH 10

2.4. Virtual Sensors and Actuators in SIDE 13

2.5. A Character Displaying Function in uCOS 17

2.6. The Process Model in PicOS 19

2.7. Inter-Process Communication in PicOS 20

2.8. A Clock Application Example 21

3.1. Compiler Segments 31

3.2. MMU Translator Blocks 34

3.3. The Emulated Memory Map 35

3.4. Spare SDRAM Emulation 37

3.5. The Port Configurator 38

3.6. The System Support Module 41

3.7. Clocking In Detail 42

3.8. CPU Clock Frequency Computation 43

3.9. Timer Frequency Computation 45

4.1. A Priority Test Application 56

List of Acronyms

ADC	Analog-to-Digital Converter
CPU	Central Processing Unit
DUART	Dual Universal Asynchronous Receiver/Transmitter
DUSART	Dual Universal Synchronous Asynchronous Receiver/Transmitter
EHI	External Host Interface
EMI	External Memory Interface
FSM	Finite State Machine
GPIO	General Purpose Input-Output
IO	Input-Output
IRQ	Interrupt Request
LCD	Liquid Crystal Display
LED	Light-Emitting Diode
MII	Media Independent Interface
MMU	Memory Management Unit
PC	Personal Computer
PCB	Process Control Block
PIO	Parallel Input-Output
PLL	Phase-Locked Loop
RTC	Run-Time Clock
SDRAM	Synchronous Dynamic Random Access Memory
SIDE	Sensors in a Distributed Environment
SMURPH	System for Modeling Unslotted Real-time Phenomena
SSM	System Support Module

1. Introduction

Advantages of emulation over other tools, such as simulation or a true physical representation, have stipulated its use in practical and academic domains. With emulation, the behavior of a system or device can be studied without interfacing to relevant hardware. A model of the system or device is developed in a different physical environment that serves as an emulating platform.

An emulated system can be useful for experimenting with the functionalities that the real system provides and for designing a suite of applications or conducting studies. This can be done in prototype mode, before gaining access to the hardware and software packages or before worrying about implementation details. In fact, an emulation platform may abstract implementation specifics away and yet at the same time provide sufficient functionality of the emulated device [Z10].

However, emulation as a research or prototype method has its limitations. If it is conducted at a low level it may limit performance of the system in a virtual setting. This may happen if it may not be real time, i.e. a unit of time elapsed in an emulated system is not equivalent to the same time unit in the actual physical system. To achieve real-time performance, the system may need to be carefully calibrated to offset the performance degradation by using a processor that is appropriately faster than the emulated processor [Z2]. In addition, as emulation becomes more architecture-specific, the results apply only to the range of devices it can successfully emulate [Z1]. At the same time, it may bring unexpected results if it allows to identify performance bottlenecks or to observe physical phenomena that are difficult to track using simulation or mathematical modeling.

Compared to emulation, mathematical models are typically an oversimplified representation of reality. They must rest on a number of assumptions that are

necessary for the analysis, and they disregard performance overheads such as context switching [Z1, Z2].

Simulation may be more successful at capturing relevant physical phenomena than mathematical models, and at the same time it is more cost-efficient than real systems [Z2]. But as compared to emulation, it is more of an abstraction (even if a more accurate abstraction) as it does not include the same level of involvement from physical devices. In contrast to simulation, emulation does not attempt to analyze performance and model the most relevant factors, but rather to give the user an impression that the emulated system is being interfaced.

Probably a better definition of emulation and simulation is given in [DIC]:

> "One system is said to emulate another when it performs in exactly the same way, though perhaps not at the same speed. A typical example would be emulation of one computer by (a program running on) another. You might use an emulation as a replacement for a system whereas you would use a simulation if you just wanted to analyse it and make predictions about it."

In this work, we present an emulation platform that is used in conjunction with a small embedded operating system. PicOS [Z5] is an operating system featuring a small footprint kernel and libraries for reactive applications. Reactive applications respond to changes in the physical environment, for instance, a computer network, in a timely manner. Unlike other embedded (sometimes called real-time) operating systems, PicOS allows an application to be structured as an explicit finite state machine (FSM) for easier development and documentation. It owes its distinct FSM design to its predecessors, SMURPH (a System for Modeling Unslotted Real-time PHenomena) and SIDE (Sensors In Distributed Environment) - packages for modeling,

simulating, and controlling systems deploying communication networks [Z4, Z6, Z7].

The current thesis work is focused on developing an IO system that is part of a larger project that aims at constructing a platform for reactive systems, primarily wireless embedded applications. In the emerging era of pervasive computing, when inexpensive ordinary devices and appliances communicate wirelessly at a new, never seen before, ubiquitous level, study of wireless platforms and network protocols has attained a research interest of imminent importance. In particular, the PicOS project targets systems that are characterized by the following features:

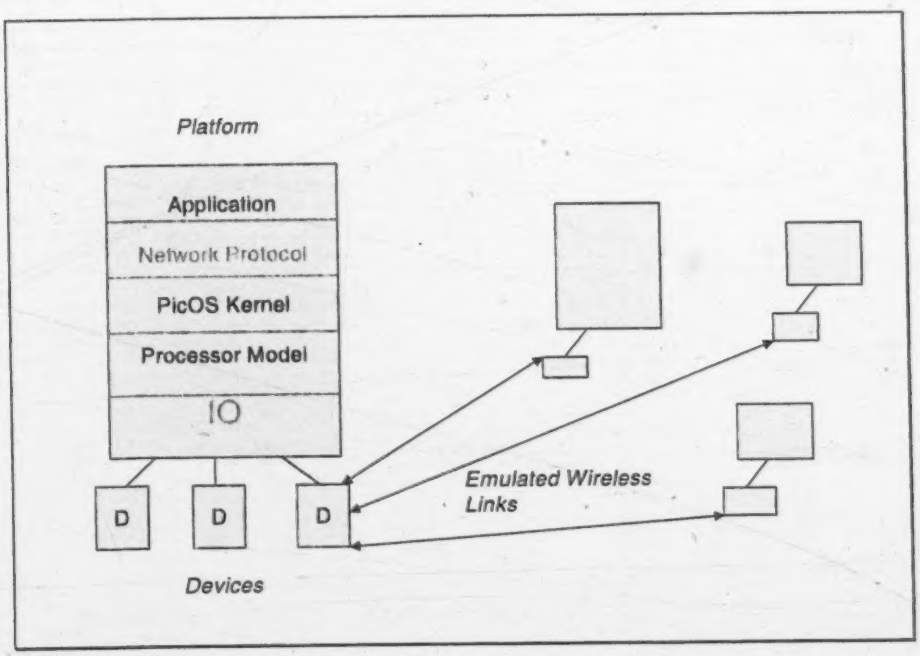
- small physical dimensions;
- small footprint;
- low-end CPU and little data memory;
- low cost;
- low power consumption;
- low-bandwidth short-range wireless devices [Z5].

Within this category of small systems, examples may include cheapest household appliances, personal use multimedia systems, remote sensors, etc.

Along with the emerging research interests in the area of small embedded network systems, there are some difficulties with regard to the research methods employed for such studies. A true physical representation of a network may be expensive. It usually involves a large number of network stations or nodes, each of which runs a version of the platform and an embedded network protocol. In addition, some physical parameters of a node, such as the (x and y) coordinates of (possibly widely dispersed) nodes may be difficult to control in a real physical setting. To this end, emulation as a method can be used to facilitate research studies, for example those targeting wireless network protocol design.

The original version of PicOS was developed for the eCOG1 microcontroller and the eCOG evaluation printed circuit board produced by the same manufacturer, Cyan Technology. To study embedded systems in an emulated environment, the PicOS project integrates a multitasking kernel for developing reactive applications, a generic transceiver interface for an easy integration of physical interfaces and protocol specifications, and an IO system for a microcontroller board, as depicted in Figure 1.1.

Figure 1.1. The Place of Emulated IO in the Platform Architecture



A network protocol can be specified in PicOS as an optional policy plug-in used by a generic transceiver (TCV) driver. A special physical interface module is used to interface the TCV driver to the IO system. The IO system is emulated in the emulated mode of operation of the platform.

The emulated IO system, which is the primary focus of this thesis, specifically includes the following elements:

- the programming interface to the microcontroller (carried out by pin connections, memory mapped registers, etc.);
- emulated semantics of IO devices;
- inter-node communication;
- PicOS device drivers or device processes (with minor deviations from those for the true physical platform), which interface the IO devices to PicOS and application processes;
- an emulated memory map of the eCOG microcontroller board;
- the IO interface at the register level (for clocks, interrupts, enable/disable operations, etc.);
- and finally, some applications are presented that test the functionality of the emulated platform.

In addition, the thesis considers how process scheduling is done in the PicOS kernel and what kernel services are offered to the application programmer. It improves on the existing process scheduling by adding a scheduling policy that is based on explicit priorities, which can be assigned to processes to achieve a better responsiveness for more urgent, time-critical, tasks (which also leads to a more optimal reactiveness of the system on the whole). The thesis describes how the new process scheduling was implemented and tested by using the platform based on the emulated IO system rather than true physical devices (to achieve a shorter debugging cycle, for example).

In section 2 we present an overview of the reactive methodology underlying PicOS design and demonstrate the programming model by using a simple example application. Section 3 describes how we emulated the peripheral devices found on the eCOG platform so that the system and applications can run in virtual conditions. Section 4 presents an example of how the virtual machine was used to develop and test a new scheduling policy for the

operating system. Finally, section 5 concludes the thesis and outlines future work.

2. Reactive Methodology

In this section, we examine a reactive methodology that is culminated in an unorthodox operating system, PicOS. PicOS is based on formalizations developed in its antecedents - software packages that model reactive systems. Although the range of applicable physical systems can be enormous, the primary domain served by the models is networking systems.

A Paradigm for Modeling Reactive Physical Systems

SMURPH, a System for Modeling Unslotted Real-time PHenomena, is a specification and simulation package that exhibits the best qualities of emulation. It was designed in such as way that it emulates real physical networks and "can be compiled into silicon" [Z4:2]. The emulation is performed via a virtual hardware platform configurable by the user. We will have a closer look at the SMURPH features that have enabled a transformation of this pure software abstraction into a system controlling real-world devices.

To emulate real systems, SMURPH is based on a process model featuring multitasking and inter-process communication and synchronization. Many running entities - tasks or processes - are interleaved to achieve an optimal usage of a shared resource that is usually available in a single copy, the processor. This interleaving is beneficial for processes performing operations of different latencies, for example input and output directed to devices of different speeds. Reactive processes, which by definition primarily respond to their physical environment, are IO-intensive, as opposed to processor-intensive, and can benefit naturally from the multitasking apparatus. Competing processes are ordered for purposes of their sequential execution, which is known as scheduling. With this technique the system user is under the impression that more than one process is executed concurrently.

SMURPH processes are specified in the object-oriented programming language supplied by the SMURPH preprocessor and a C++ compiler. A process skeleton looks as follows:

Figure 2.1. The Process Model in SMURPH [Z4]

```
process ptype: supptype (fptype, stype) {
    ... attributes and methods ...
    setup (...) {
      ...
    };
    states {s0, s1, ... , sk} ;
    perform {
      ...
    };
};
```

where *ptype* stands for the process type, *supptype* is a previously defined process type, *fptype* is the type of the process' parent, and *stype* is the type of the station owning the process [Z4:9].

A process is structured as a modified C++ class. It includes process data represented as a number of so-called process attributes. Unlike a PicOS process, it can also include sub-functions, or in object-oriented terminology, methods, directly in its definition. These methods can be specified as regular C++ methods. The optional setup method is similar to an object constructor. It is invoked to initialize object data attributes. The perform method in each SMURPH process is the process code function. It is modeled around explicitly declared process states. The states may be used to formalize the process' behavior in response to its environment.

The process environment is modeled in SMURPH via simulated networking devices. It consists of the station owning the process and other variables among which is the current packet, *ThePacket*, and the current port, *ThePort*. In this environment events are triggered that may be awaited by a process for purposes dictated by the application logic. The occurrence of an event forces

the process into a specific state. As networking application processes exhibit reactive nature they typically wait for external events triggered by a networking device and do little processing. This observation is the foundation for event-driven networking modeling tools, as well as the PicOS process model that exploits the reactive nature of embedded applications to its potential.

The process code method in SMURPH formalizes the finite state machine of a particular process [Z4:10]:

Figure 2.2. The Perform Method in SMURPH [Z4]

```
perform {
    state s0:
      ...
    state s1:
      ...
    ...
    state sk:
      ...
} ;
```

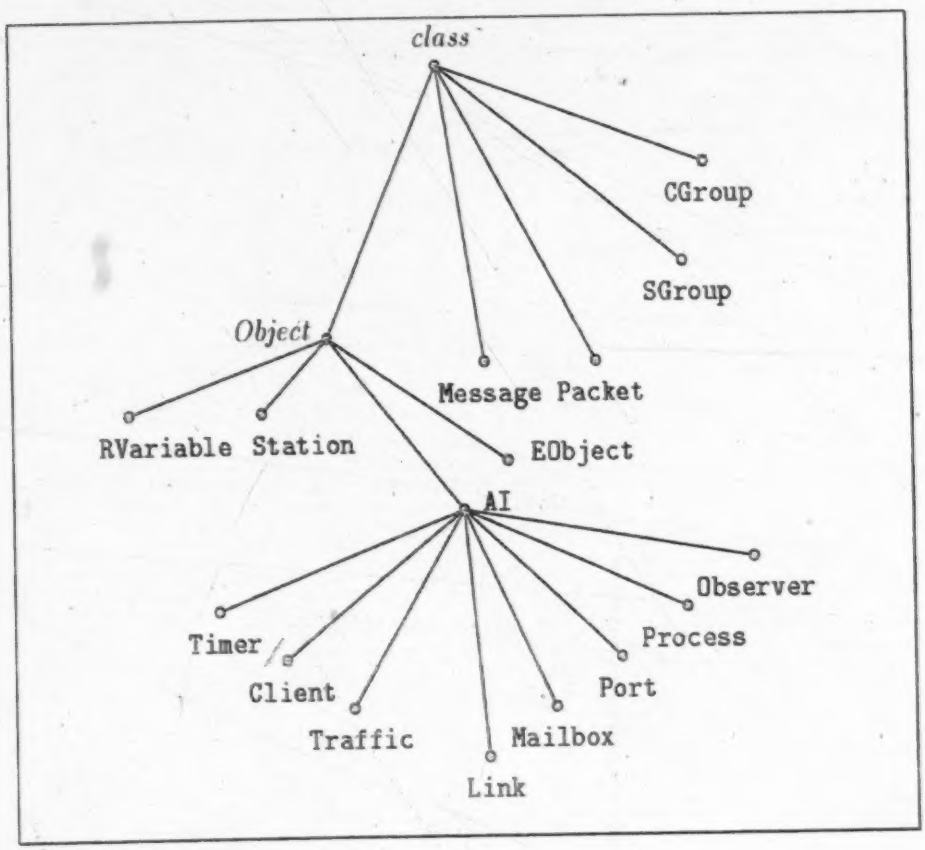
The instra-state code may include SMURPH-defined function calls that wait for events.

The physical environment is represented to the kernel and processes by special objects called Activity Interpreters. Events that are awaited by processes are AI-specific. For instance, a wait call, such as *ai->wait (event, state)*, will put the caller process to sleep waiting for the named event that is triggered by the named Activity Interpreter. The process will be resumed in the named state. All AI wait requests issued by a running process combine into alternative waiting conditions. As soon as any of them is satisfied, the waiting status is cleared and all other (unsatisfied) wait requests are erased.

Activity Interpreters, through which the world outside the control program is modeled, include the timer, traffic objects, IO ports, mailbox objects (for inter-

process communication), and processes (for orthogonal synchronization) (see Figure 2.3.). As such, they trigger events external to the process and advance the system time by an amount appropriate for the physical phenomenon they represent. For example, a process calling the wait method of a mailbox, such as *mb->wait (NEWITEM, gotit)*, will signal that it is waiting for a NEWITEM event in the mailbox *mb*. Upon the occurrence of this event, the process will find itself in the state labeled *gotit*. The mailbox *mb* is responsible for updating simulation time by a value determined by *mb*.

Figure 2.3. The User-Visible Class Hierarchy in SMURPH [Z4:5]



To synchronize with another process, a process may issue a wait call on that process which acts as an Activity Interpreter for the first process. *p2-> wait (p2stateA, p1stateB)* will make the caller process *p1* wait until the process *p2* is in state *p2stateA* and then transit to state *p1stateB*. In another example,

```
prc = create mychild ( ... ) ;
prc->wait (DEATH, done);
sleep;
```

the calling process is synchronized with its child. The parent creates the child and waits for its completion before it proceeds any further [Z12:23-24].

While a process is sleeping the simulated time flows in terms of indivisible time units. Events happened during each indivisible time unit are not ordered deterministically. When a process is running time does not flow, as time is advanced by Activity Interpreters only. This is the nature of discrete-time simulation in SMURPH.

As a simulator, SMURPH collects performance (quantitative) and correctness (qualitative) data, and can be used to evaluate different Medium Access Control protocols. In addition to collecting standard performance measures it monitors process execution using so-called observers. Observers is a methodology for testing protocol conformance via dynamic process-like assertions. These assertions run over a number of other (regular) processes. For example, the inspect operation, *inspect (s, p, ps, os)* will restart the observer in state *os* "when process of type *p* owned by station *s* is awakened in state *ps*" [Z4:13]. The call *inspect (ANY, ANY, ANY, os)* will apply to processes of any type that are owned by any station and executing in any state [Z4:13].

A feature that draws SMURPH closer to systems controlling physical devices is its configurability. For each particular application, it is configured into a specification model of a particular protocol. The mechanism by which the platform is used by an application is natural in the world of embedded systems.

User-provided code resides in a number of source files. It makes use of the SMURPH library of functions and definitions. A program called mks, whose purpose is similar to that of the make utility, is made available for compiling and linking select source and library files into a stand-alone application.

In conclusion, SMURPH is structured as if it runs on a hypothetical hardware that it effectively emulates. Unlike other simulation packages, it has a built-in model for predicting event timing and is able to schedule these events [Z12:3].

A Control Specification for Distributed Devices

SIDE (Sensors in a Distributed Environment) [Z6] is an extended version of its predecessor, SMURPH, that can be used to control sensor and actuator devices over the Internet. The range of the devices that are controllable by a SIDE program is represented by distributed Internet-accessible devices. The execution of SIDE processes can be monitored and operated via a Java applet from a Java-capable web browser.

Physical sensors and actuators are seen in the control program as abstract virtual sensors and actuators. In this way, the control program is not dependent on the presence and physical characteristics of a particular device. On the contrary, it can operate in a purely virtual environment, in the so-called virtual mode (when all devices in the system are virtual). This mode is different from the real mode, in which configured devices may be physical. In this case, the virtual sensors and actuators are translated by a separate software layer into their physical counterparts. This correspondence is flexible and adjustable. For example, a logical (SIDE) sensor or actuator can be remapped to a different, or a set of, physical devices.

It its virtual mode, SIDE acts as a discrete-time simulator, and no physical devices may be present in the system. In the real mode, physical sensors trigger

events in real-time and may affect the behavior of physical actuators. Some devices may still be virtual, but this is transparent to the application.

The control program and the kernel run on a typical operating system, e.g. Linux. The outside world in SIDE is represented by software abstractions - mailboxes - that can be mapped to TCP/IP ports and some peripheral devices. The ports are managed by daemons, i.e. processes that carry out the translation between the physical devices and logical abstractions, mailboxes in this case. They can be compared with operating system device drivers that interface physical devices to operating system abstractions through io system calls.

Virtual sensors and actuators are derived from the Mailbox class [Z11:5]:

Figure 2.4. Virtual Sensors and Actuators in SIDE [Z11]

```
mailbox Sensor {
private:
    int Value;
    void mapNet ();
public:
    NetAddress Reference;
    void setValue (int);
    int getValue ();
    void setup (NetAddress&);
};

Mailbox Actuator : Sensor { } ;
```

The value in a mailbox is used differently by a sensor and an actuator. For a sensor, it represents the perceived environment. For an actuator, it is the description of an action to be performed by the actuator.

The mailbox acts as the buffer in a high-level device driver, here a daemon. The virtual devices act as consumers of the IO operations, similar to the application-driven IO system calls, sensors for reading and actuators for writing. For example, when an appropriate number of bytes is received in the

mailbox, the mailbox triggers an event to awake a waiting reader. The reader may synchronize with the IO operation in three different ways. It can "suspend itself awaiting data arrival (the conventional approach), ignore the operation and poll the mailbox later (non-blocking I/O), or spawn a separate process to read the data when it becomes available and notify its parent about that fact via a signal" [Z11:19].

The SIDE, as well as PicOS, kernels are non-preemptive so that the reactive input/output operations are easy to synchronize. Processes may be preempted at state boundaries but intra-state code runs to completion without interruptions. This is one of the distinctive features of the PicOS embedded system inherent in its reactive nature. In the next section, we examine the state of the art in modern embedded operating systems to highlight the place of PicOS' reactive paradigm in it.

Contrasts Between Preemptive Systems and PicOS

A typical embedded operating system is preemptive, a quality that allows it to be called real-time. At almost any point in time a running process may be preempted by another process if the scheduling priority of the latter is higher. In order to resume a preempted process the system must save its context in a stack memory area that is process-specific. In other words, the stack pool is divided into disjoint private process stacks identifiable by each process control block. As the stack memory is fragmented among the processes, the number of nested function calls and local function variables and parameters in a process may be severely limited on the typical memory-constrained microcontroller platform. The programmer is forced to come up with an evaluation (at times, too conservative or too optimistic) of stack requirements for each created process.

In PicOS all of the system stack memory is available to a process, and all processes execute from the same physical address. The reuse of the same stack space is made possible by co-operative non-preemptive scheduling controlled by the process. The process designer is free to control the CPU whenever it is allocated to the process. If intra-state code is not long (and it is typically short for IO-bound systems) the possible extra execution time attributed to a running process (whenever a higher priority task is ready) may effectively offset overheads of a preemptive system such as the expensive context switches. A PicOS process identifies when it should be preempted. This is in contrast to the prevailing practice of identifying when a process should not be preempted.

In a preemptive system, the application designer should expect preemption of a process at any instruction or statement and guard critical sections of code, provide mutual exclusion, protect against race conditions, etc. A failure to do so may result in hard-to-find errors. Critical sections of code are guarded by operations that involve disabling and enabling all interrupts. Whereas PicOS may disable interrupts on a few occasions in low-level IO functions, preemptive systems need to use the trick more liberally, to protect software process synchronization constructs. For example, the frequently executed scheduler may disable all interrupts to prevent an interrupt service routine from introducing a new high priority ready task while it is being determined by the scheduler. Disabling interrupts increases interrupt latency and may result in missed interrupts.

When a preemptible task is created, along with code and data pointers and a task priority, a stack pointer must be specified. The stack size and location are constrained by a number of factors: the available RAM space, the number and size of stacks for other processes, the anticipated function and interrupt nesting. Whereas the programmer may have some control over the task stack requirements, such as those dictated by the number and size of local variables and nested function calls, interrupt requirements are harder to predict. Interrupts may execute in the running task context, and in some systems may

be nested. CPU register values and other run-time context must be saved on the same stack during interrupts. A context switch is costly. Besides saving context on the stack, it passes control over to the new running task. In a preemptive system, a context switch may be modeled similar to interrupt processing and uses a similar algorithm [U2, U3].

The memory required by a process control block in PicOS with standard settings for configurable parameters is 20 bytes [Z5]. uCOS (micro-Controller Operating System), an open-source preemptive microcontroller operating system which is representative of its class, requires a stack for each task. Its memory requirements are 10 bytes for a task control block and a typical 256 bytes for a stack [U2], 266 bytes in total. The improvement in RAM usage offered by PicOS is thus more than tenfold. With the stack size of 512 bytes used by PicOS the improvement for one task would be more substantial. It certainly accumulates as the number of tasks increases. It is typical for an embedded system of this size to have up to 16-32 tasks.

Synchronization in Preemptive Systems

Semaphores, mailboxes, queues, etc. are various software constructs implemented in terms of lower-level atomic elements that guarantee that a piece of code is executed contiguously as one unit, without interruptions. This contiguous piece of code is sometimes referred to as a critical section. The atomic operations are guaranteed by hardware - for instance, interrupt disabling or a special Test-and-Set instruction. The software synchronization constructs use the atomic operations in an optimized fashion so that those operations are in effect for a minimum amount of time. For example, they are applied to check or update the count variable in a semaphore or the message buffer in a mailbox.

The synchronization mechanisms are costly because they:

- serialize access to a shared resource by disabling all interrupts;
- usually give rise to elaborate and error-prone synchronization algorithms;
- occupy code and data memory space;
- if the resource is busy and no timeout mechanism is used the process may block indefinitely if no remedy solutions are introduced.

The ROM requirements for the implementation of synchronization constructs in the uCOS operating system are quoted as follows [U3]:

Semaphores	500 bytes
Mailboxes	500 bytes
Queues	600 bytes

Altogether, the three synchronization services occupy more than half of the 3K byte kernel [U3].

The data memory for the same constructs is 13 to 26 bytes each. A typical application may have on the order of 10 semaphores, 5 mailboxes, and 5 queues of 10 entries. Each message in a queue may need 2 bytes [U1]. This configuration of synchronization primitives results in about 425 bytes of additional RAM requirements. In addition, the synchronization operations for pending and posting to semaphores, mailboxes, and queues, disable interrupts for 400 CPU clock cycles each [U1].

Synchronization constructs are defined as global variables and used for accessing shared resources such as IO devices and inter-process communication buffers. By way of example, a character displaying function may look as follows:

Figure 2.5. A Character Displaying Function in uCOS [U1]

```
void DispChar (UBYTE x, UBYTE y, char c) {
```

```
  UBYTE err;

  OSSemPend (DispSem, 0, &err);
  gotoxy (x+1, y+1);
  putchar (c);
  OSSemPost (DispSem);

}
```

The *putchar ()* function is not reentrant, i.e. at most one process instance may execute it at any given time. Therefore, its caller must guard the printing to the display. The *DispSem* semaphore is initialized to 1 to indicate that only one process can access it at a time.

To sum up, the timing of task preemption in a preemptive system is non-deterministic. In PicOS the application designer has the freedom of structuring a process around explicit preemption points depending on the process needs to achieve a more optimal interleaving of operations of different latencies, such as IO and general processing. This is especially beneficial for applications of a reactive nature, i.e. those responding to events in their environment. The perception of and response to the environment are performed via peripheral devices.

Kernel Services for Reactive Applications

The reactiveness of PicOS applications is manifested through the structure of a process and kernel services explicitly operating on processes and process states.

A PicOS process is defined using PicOS-specific keywords: *process, entry, endprocess, nodata,* as in the following model:

Figure 2.6. The Process Model in PicOS

```
#define STATE0    0
#define STATE1    10
  . . .
process (process name, process data type)
      static local variable declarations; // static declarations will survive state
transitions

   entry (STATE0)
      intra-state code;

   entry (STATE1)
      intra-state code;

endprocess (maximum number of process instances)
```

This programming model is illustrated in the following sections with examples.

Processes are created by the *fork()* operation. The first, and main, application task must be called root. It is created by the system automatically. A process may terminate itself or be killed by another process.

Perhaps the most distinctive services are those for process coordination. A process may wait for the occurrence of an event and then transit to another state. It explicitly controls whether and when it may be preempted by the scheduler for another process (by the *release* operation). A process may signal, or trigger, events that are perceived by all other processes in the system. For instance, an IO read system call may not return data immediately. The issuing process will be put to sleep awaiting an event signaling that data is received. When data is available for reading, the IO driver will trigger the data ready event that will wake up the sleeping process.

If the event waited for is a timer event then the delay operation is used. It is similar to wait in that the caller specifies an event (in this case the duration of a time delay) and a state to transit to upon occurrence of such an event. In

addition, there are other variants of the delay function that block the caller for a specified number of milliseconds or microseconds.

Process functions may be reentrant or not. Again, this is controlled by the application designer by specifying the parameter value of the *endprocess* macro. If it is 1 then the process code may have only one corresponding process instance, i.e. it is not reentrant. If it is more than 1 then the maximum number of process instances for the reentrant code function is specified. *endprocess (0)* stands for no explicit limit on the degree of multiprogramming.

As PicOS processes are not preemptive in their intra-state code, they can easily communicate and synchronize using simple global data structures. Alternatively they can be created on the same data object and communicate or synchronize through it, as in the following example:

Figure 2.7. Inter-Process Communication in PicOS [Z5]

```
process (coordinator, bufstr_t)

    entry (MN_START)

        data->mon = fork (monitor, data->buf);
        for (int i = 0; i < data->N; i++) {
            fork (consumer, data->buf);
            fork (producer, data->buf);
        }

    entry (MN_WAIT)

        if (!running (consumer) || !running (producer)) {
            killall (producer);
            killall (consumer);
            kill (data->mon);
            terminate;
        }

        joinall (consumer, MN_WAIT);
        joinall (producer, MN_WAIT);

endprocess (1)
```

In this example the three child processes, monitor, consumer, and producer are created on the same *data->buf* object. *Data* is the default data object in the current process, the coordinator process in this case. Unlike local process variables (if not declared static), the data variable survives state transitions that involve the scheduler.

Using PicOS as a Platform for Embedded Applications

The PicOS platform includes a multitasking kernel and a library.

The kernel provides standard operations for creating processes, IO operations and device drivers, user-defined countdown timers and delay functions, dynamic memory allocation and deallocation, string operations. In addition, specific functions are provided that support the FSM structure of a process and event-based synchronization. A process is ready for execution if it is not waiting for any of the events, such as the expiration of a timer (signaled by the timer service), completion of an IO operation (signaled by a device driver), termination of a process (signaled by a killing process), etc. Further, applications are supported by a library of IO and formatting string functions that can be called from an application and generally not used by the kernel.

Consider the following application example that demonstrates the timing capabilities provided by the kernel. Specifically, it illustrates a periodic triggering mechanism that can be used, for example, in a simple alarm clock.

Figure 2.8. A Clock Application Example [Author: E. Akhmetshina]

```
#include "sysio.h"

heapmem {100};
```

```
#include "lcd.h"
#include "ser.h"
#include "form.h"
#include "beeper.h"

unsigned short hr=0, min=0;
char *fm;
unsigned short amin=0;
unsigned short btone=6;
char *ibuf = NULL, *obuf=NULL;

#define RS_MIN  00
static process (clock, void)

  nodata;

 entry (RS_MIN)

      delay ( /*1024* */60, RS_MIN+1);      //timer delay
      release;  //pass control to the scheduler

 entry (RS_MIN+1)

     min++;   //update minutes
     if (min==60) {
        min = 0;
        hr++;  //update hours
     }

     if (hr==24) {
        hr = 0;  //update hours
     }

     form (fm, "%u:%u", hr, min);    //format a string
     upd_lcd (fm);     //update the LCD
     proceed (RS_MIN);       //proceed to a different state

endprocess (1)
#undef RC_MIN


#define RS_AL  00
static process (alarm, void)

      static word adelay;
      nodata;
```

```
entry (RS_AL)

    obuf = "Alarm will go every %u seconds\r\n";
    adelay =/*1024* */  60* amin ;

entry (RS_AL+1)

    ser_outf (RS_AL+1, obuf, adelay);  //serially output a formatted string

entry (RS_AL+4)

    delay (adelay, RS_AL+2);        //timer delay
    release;              //release to the scheduler

entry (RS_AL+2)

    diag ("buzz");      //diagnostic serial output
    beep (1/1024, btone, RS_AL+4);          //sound the beeper

entry (RS_AL+3)
    finish;     //end the process

endprocess (1)
#undef RC_AL


int clock_start (void) {

     if (! running (clock)) {
        fork (clock, NULL);   //create new process
        return 1;
     }
        return 0;
 }

void clock_stop (void) {

      if (running (clock)) {
         kill (running (clock));  //terminate process
      }
}

void alarm_stop (void) {

      if (running (alarm)) {
         kill (running (alarm)); //terminate process
```

```
    }
}

int alarm_start (void) {

    alarm_stop();
    fork (alarm, NULL);  //create new process
    return 1;
}


#define RS_INIT     00
#define RS_RCMD     10
#define RS_CLK      20
#define RS_ALR      30
#define RS_KC       40
#define RS_KA       50
#define RS_BEP      60


#define BUFSIZE     128

process (root, void)    //main process in an application

 nodata;

 entry (RS_INIT)

    dsp_lcd ("PicOS ready    ALARMCLOCK", YES); //display on the LCD

    if (ibuf == NULL)
        ibuf = (char*) umalloc (BUFSIZE); //dynamically allocate memory
     if (obuf == NULL)
        obuf = (char*) umalloc (BUFSIZE);
     fm = (char*) umalloc (6);

     min = (unsigned short) seconds () / 60;
     hr = min / 60;
     min = min % 60;

     proceed (RS_CLK+2);  //proceed to a state

 entry (RS_RCMD-1)

     ser_out (RS_RCMD-1,
      "\r\nAlarm clock\r\n"
```

```
    "Commands:\r\n"
    "c hour minute        (clock time)\r\n"
    "a minutes            (alarm frequency)\r\n"
    "q                    (quit)\r\n"
    "s                    (stop alarm)\r\n"
    "b tone               (beeper tone)\r\n"
    );          //serially output (to the current DUART)

entry (RS_RCMD)

    ser_in (RS_RCMD, ibuf, BUFSIZE);    //serially input

    if (ibuf [0] == 'c')
        proceed (RS_CLK); //proceed to a state
    if (ibuf [0] == 'a')
        proceed (RS_ALR);
    if (ibuf [0] == 's')
        proceed (RS_KA);
    if (ibuf [0] == 'q')
        proceed (RS_KC);
    if (ibuf [0] == 'b')
        proceed (RS_BEP);

entry (RS_RCMD+1)

    ser_out (RS_RCMD+1, "Illegal command\r\n"); //serially output
    proceed (RS_RCMD-1);

entry (RS_CLK)

    if (scan (ibuf + 1, "%u %u", &hr, &min) < 1 || hr > 24 || min > 60)
        proceed (RS_RCMD+1);

entry (RS_CLK+2)

    form (fm, "%u:%u", hr, min);    //form a string
    dsp_lcd (fm, YES);        //display on the LCD

    if (clock_start ())
        obuf = "Started clock, hours = %d, minutes = %d\r\n";
    else
        obuf = "New clock parameters, hours = %d, minutes = %d\r\n";

entry (RS_CLK+1)
```

```
    ser_outf (RS_CLK+1, obuf, hr, min); //serially output a formatted string
    proceed (RS_RCMD-1);

entry (RS_ALR)

    if (scan (ibuf + 1, "%u", &amin) < 1 || amin > 60 || amin ==0)
        proceed (RS_RCMD+1);

    if (! running (clock)) {
        proceed (RS_RCMD+1);
     }

     if (alarm_start ())
         obuf = "Started alarm, minutes = %d\r\n";
     else
         obuf = "New alarm parameters, minutes = %d\r\n";

entry (RS_ALR+1)

    ser_outf (RS_ALR+1, obuf, amin); //serially output a formatted string
    proceed (RS_RCMD);

entry (RS_KA)

    alarm_stop ();
    proceed (RS_RCMD);

entry (RS_KC)

    alarm_stop ();
    clock_stop ();
    proceed (RS_RCMD-1);

entry (RS_BEP)

    if (scan (ibuf + 1, "%u", &btone) < 1 || btone > 6 || btone <1 )
        proceed (RS_RCMD+1);
    proceed (RS_RCMD);

endprocess (1)
```

Each process in PicOS is structured as a finite state machine. The *root* process, which is the main application process in a project, has several states in this example. The states are defined as literal integers to which symbolic names can

be attached for user-friendly identification. For instance, the initial state 0 is identified as *RS_INIT* in *root*. In this state the process displays a string on the LCD. For the *dsp_lcd* library function, the second argument tells whether the function should stop writing out a previous message, if any. String buffers are dynamically allocated using *umalloc()*.

Before the user sets the current hour and minute, the process obtains the number of seconds elapsed since the system startup by calling the kernel operation *seconds()* and initializes the clock displayed on the LCD. Current time is displayed on the LCD in state *RS_CLK+2* which is *proceed()*'ed to from state *RS_INIT*. *clock_start()* is an example of an ordinary C function that *fork()*'s a PicOS process (the clock process in this case).

When the clock process is created, no process data is passed through the second argument declared void (also corresponding to the *nodata* declaration). To update the LCD at a minute interval, the process uses the *delay()* kernel operation. After the number of milliseconds (a millisecond is defined as 1/1024 of a second) specified in the first argument the process will be awakened in the state specified in the second argument. Note that in this example, the minutes are 1024 times faster, for illustration purposes. (Standard C comments can be embedded in the PicOS programming language.) At the end of each minute the clock process finds itself in state *RS_MIN+1* where the current hour and minute are calculated and updated on the LCD using *upd_lcd()*.

After creating a clock process, *root* falls through state *RS_CLK+1* to write the output string *obuf* to the currently chosen DUART. It possibly loops in the same state if the IO operation can not complete immediately. When the string is written out, the process proceeds to state *RS_RCMD-1* to output command options to the user and read the response in state *RS_RCMD*.

If the user chooses to set the clock, the current time is updated on the LCD by setting new clock parameters. Alternatively, the use can set the alarm to go off

with a certain frequency in minutes. The alarm process is started in the *alarm_start()* function, which is similar to *clock_start()*. The process uses the same *delay()* operation to transit in a calculated number of milliseconds to state *RS_AL+2*, in which the beeper sounds a user-settable tone. The processes can also be explicitly killed when the user decides to do so.

An analysis of the alarm state machine may reveal that state *RS_AL+3* is never reached and is there for demonstration only. Also note that the states do not have to be kept sorted and can be inserted anywhere in the list. The initial state should be 0.

With this overview of the paradigm and the programming model offered by PicOS we proceed to a description of the virtual machine that has been instrumental in developing PicOS applications and services. In addition, in Section 4 the reader may find further information on the programming model and more examples.

3. The Virtual Machine

Introduction

The emulation undertaking we describe in this section may be classified as similar to a machine simulator of a MIPS multiprocessor in its emulation mode [Z5] but applied to a microcontroller domain. Their SimOS machine simulator is used to virtually run and analyze a general-purpose operating system (IRIX) and applications (such as the SUIF research compiler) [Z5, Z8, Z9]. The emulation mode is used as a means of overcoming the slow simulation speed of the software. For instance, "I/O devices such as disks are configured to instantaneously satisfy all requests, avoiding the time that would be required to simulate the execution of the operating system's idle loop" [Z5:85]. Altogether, "the only requirement is to correctly execute the workload; no statistics on workload execution are required" [Z5:85]. This mode may not be as useful as a simulation mode, for example for computer architects, but may provide a valuable prototyping platform for system and application programmers or a tool for application users. It may also provide "a second opinion" [Z10] to help discriminate between a software bug and a malfunctioning device.

Our goals were similar to those in a machine simulation study [Z10]. Specifically, we wanted a model that would:

- Execute the IO as quickly as possible.
- Be as realistic as possible (by emulating real-world microcontroller devices).
- Provide a clean user interface and control input and output (possibly interleaving) from different devices.
- Have a short startup time.
- Reuse as much of the functionality in the microcontroller emulation, compiler features, OS device drivers, as possible.

Constituents

The following parts constitute the virtual machine.

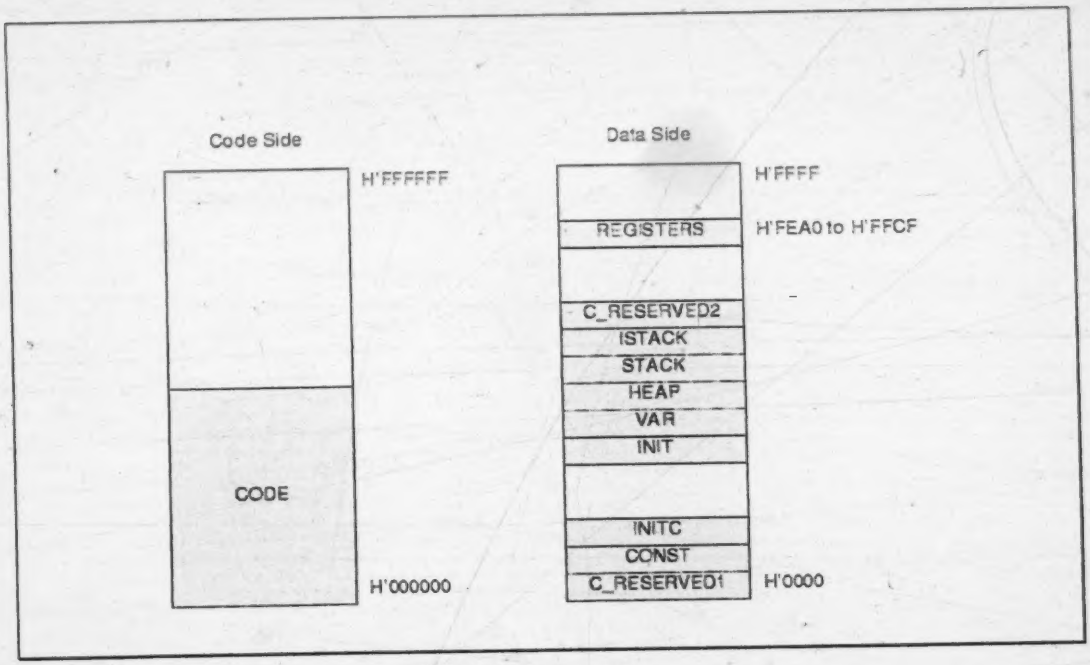
ecogsim Custom DLL

ecogsim, an emulator package that is part of the eCOG evaluation board toolkit, allows integration of an IO emulation system into the so-called "simulated" (or rather emulated) processor core. The custom IO system is written as a dynamically linked library (DLL). For ranges of addresses specified as custom, reading and writing is done by the customization module allowing us to model any devices other than the processor core.

rg

The C compiler relies on the availability of different segments (such as VAR, STACK, HEAP, etc.) [ECM:40], among which is REGISTERS (see Figure 3.1.). It is allocated for hardware registers and located in the logical address range 0xFEA0 to 0xFFCF. A variable called *rg* is reserved at this location by the startup code so that the registers can be accessed through this variable in plain C language. The emulation benefits from reading *(core_read_mem)* and writing *(core_write_mem)* to the core emulation in *ecogsim* to query or keep track of register values. This is done in addition to directly manipulating custom-mapped registers and reusing the register (*rg*), register field (*fd*), and mask definitions (from the header file *ecog1.h* provided by the toolkit).

Figure 3.1. Compiler Segments [ECM:40]



PicOS Device Drivers

The IO emulation is powered by PicOS device drivers for the eCOG evaluation board. The device drivers include those for Ethernet, radio, DUARTs, LCD, LEDs, SDRAM. The virtual machine mimics the operation of these and other devices (without a traditional OS driver), such as the clock, the MMU, etc. They can also be abstracted as devices at a higher level of decomposition. With or without a formal driver, a device may need to be provided with common functionalities, such as initialization, read, write, and the interrupt service routine. Additionally, some device data, such as queues and buffers,

must be maintained. When addressing the different devices, we highlight these issues and present an introduction to the related hardware operation and configuration, as well as the virtual operation, or emulation.

MMU and Memory Devices

Introduction

The Memory Management Unit, the MMU, carries out the translation between logical and physical address spaces. The physical memory devices are mapped to different locations in the logical address space so that programs can operate on a single, uniform, and linear address space. Thus, the purpose of the MMU is to act as an intermediary between the CPU and all the memory devices.

The eCOG follows the Harvard architecture and has a separate logical data and logical program address space. The MMU consists of 13 translator blocks, four of which are for code and nine for data translations. They are listed in the order of precedence as follows [EHM:38]:

"Code Address Translation:

1. Flash (default translation at Reset)
2. RAM (read only)
3. External Chip select 0 (CS0) memory (Read only)
4. External Chip Select 1 (CS1) memory (Read only)

Data Address Translation:

1. Internal IO register space (fixed mapping at top of address space)
2. First RAM (default translation at Reset)
3. Second RAM translator
4. Flash
5. Cache memory bank 0
6. Cache memory bank 1
7. First external Chip Select 0 memory translator
8. First external Chip Select 1 memory translator
9. Second external Chip Select 0 memory translator
10. Second external Chip Select 1 memory translator"

The order of precedence matters when different physical locations overlap, i.e. are mapped to the same logical address(es). Moreover, code and data sections may map to the same physical locations, in which case the difference in access privileges between the two blur for the address range.

Figure 3.2 [EHM:39] shows in detail the MMU translators and physical memories that each of them can access. Each translator is instrumented with a logical address register, a physical address register, and a size register to carry out the mapping [EHM:44]. The registers are set when the translator is disabled, after which the translator is enabled.

Virtual Operation

Both *ecogemu* (the loader/debugger for the actual board) and *ecogsim* (the core emulator) have built-in "boot loaders" (simple "device drivers") for the permanently-mapped (to code) internal flash. They are invoked "behind the scenes". However, the toolkit contains code example for boot loaders for the internal flash, external flash, and ram. Before a flash programmer application can start, the MMU must be preconfigured by using a batch file. For the internal flash programmer application, the MMU is set up to map three sections only: code RAM, data RAM, and registers, needed to run the application itself. The memory is programmed in units of pages by using the eICE interface to write the eCOG flash registers.

ecogsim does not model the MMU, perhaps except that the respective register (*rg*) variables retain their values at any time, and works with logical addresses only. The *app.cmd* file maps some register addresses as *custom*, and other logical locations as *rom* or *ram*. The memory map (from *app.cmd*) looks as follows.

Figure 3.2. MMU Translator Blocks [EHM:39]

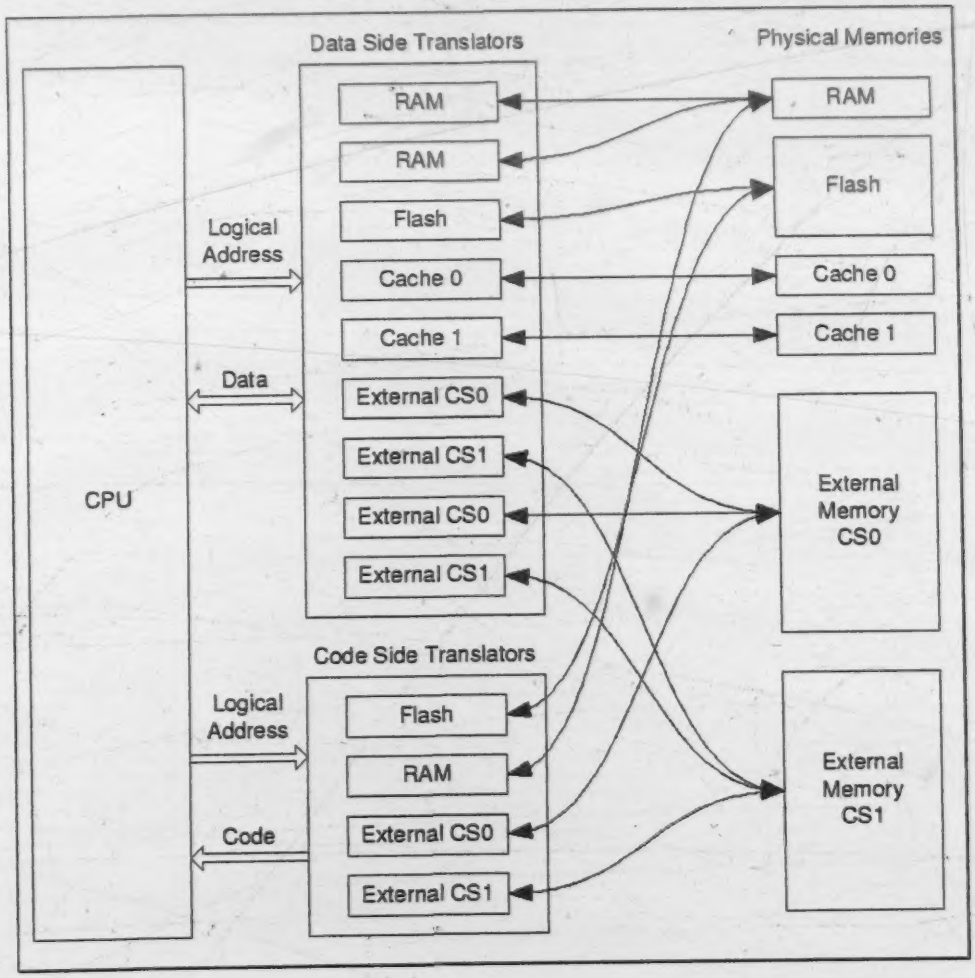


Figure 3.3. The Emulated Memory Map [Author: E. Akhmetshina]

```
> reset ALL

Install the Simulator IO System.
> install_custom "..\..\simnets\debug\simnet.dll"

Configure the memory map.
> map_log 0 7ff rom
> map_log e800 efff ram
> map_log p'0 p'77ff rom

CACHE in cstartup.asm
> map_log 1000 1400 ram

sdram (need be custom as physical spare SDRAM >32K in logical data space)
map_log 4000 bfff custom
>map_log 4000 bfff ram
SDRAM: mmu.ext_cs0_data[0-1]_[log,phy,size] read from coremem if needed

-------map some IO registers as custom------

LEDS: io.gp0_3_out
> map_log ffad custom

DUART: duart.[a-b]_[tx8,tx16,rx]
> map_log fea8 feaa custom
> map_log feb1 feb3 custom
DUART: duart.[a-b]_sts, duart.[a-b]_int_[dis,en]
> map_log fea4 fea6 custom
> map_log fead feaf custom

LCD: io.gp8_11_out
> map_log ffaf custom
LCD for speed: gp16_19_out, gp20_23_out
> map_log FFB1 FFB2 custom

CLOCK: ssm.cpu, tim.int_en1, tim.int_dis1
> map_log ff72 custom
> map_log ff3b custom
> map_log ff3d custom
CLOCK support for RADIO XEMICS: tim.ctrl_[en,dis] -- to keep track of
CNT1 on/off as a proxy for Txing/Rxing
> map_log ff1e ff1f custom
```

```
ETHER: {ADDR[0-2]_REG, DATA_REG, RXFIFO_REG} and
MMU_CMD_REG
> map_log 3d82 3d84 custom
> map_log 3d80 custom
the other ETHER registers must be mapped as well to avoid trap write_none
> map_log 3d81 ram
> map_log 3d85 3d87 ram

BEEPER: ssm.tap_sel2, ssm.rst_set
> map_log ff77 custom
> map_log ff6a custom

RADIO: RFMI or VALERT: io.gp8_15_sts (Rxing), and rg.io.gp12_15_out
(Txing)
> map_log ffb6 custom
> map_log ffb0 custom
RADIO: XEMICS: io.gp0_3_out (Txing, o/w LEDs, @ FFAD), io.gp8_15_sts
(Rxing)
+ for speed:
> map_log ffaa custom

turn off READ_X exception (read from unintialized data location)
> trap READ_X OFF

----------

Load the eCOG code.
> load "image"

> go
```

PicOS maps the first 32K of SDRAM starting at logical address 0x4000. The rest of the available SDRAM can be dynamically mapped in and out of that block. A prototype implementation of the virtual machine mapped the address range 0x4000 to 0xbfff as custom. Based on the value stored in the rg.mmu.ext_cs0_data0_phy register, it directed a SDRAM memory reference to the appropriate location in the emulated full-size SDRAM, as follows:

Figure 3.4. Spare SDRAM Emulation [Author: E.Akhmetshina]

```
coremem ssdram [SDRAM_NBLOCKS] [SDRAM_BLKSIZE];
#define inSDRAM(adr) ( (adr >=SDRAM_ADDR) && (adr
<SDRAM_ADDR+SDRAM_BLKSIZE) )

coremem SDRAMr (coreadr adr) {
      coremem blk;

      if ( inSDRAM(adr) ) {
        core_read_mem (rg2addr(rg.mmu.ext_cs0_data0_phy), &blk);
        return ssdram [blk][adr-SDRAM_ADDR];
       }
}

int SDRAMw (coreadr adr, coremem val) {
      coremem blk;

      if ( inSDRAM(adr) ) {
        core_read_mem (rg2addr(rg.mmu.ext_cs0_data0_phy), &blk);
        ssdram [blk][adr-SDRAM_ADDR]= val;
        return SUCCESS;
       }
}
```

However, this solution had to be rejected. When SDRAM was mapped as *custom*, any memory reference to SDRAM (even to the first 32 K block, for example, when dynamically allocating memory using malloc()), slowed down the emulated system significantly. The current solution is to model the 32K block as *ram* and forgo the modeling of spare SDRAM in order to benefit from improved performance.

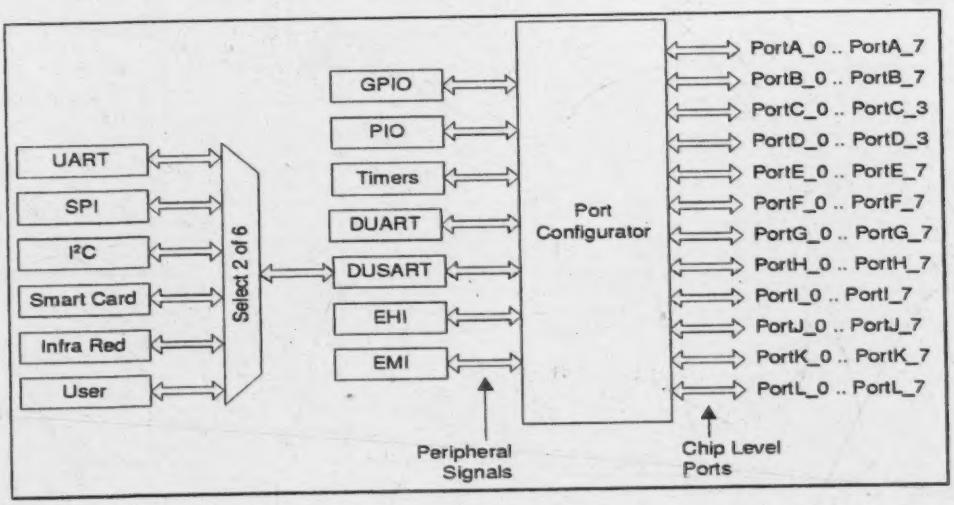
Another EMI memory-mapped physical device is the Ethernet chip. Unlike SDRAM, it uses the bus EMI format [EHM]. The virtual machine simply maps the Ethernet memory space as *custom* (for some memory-mapped registers) or *ram* (for others).

CPU's Peripheral IO Ports

Port Configuration

The available twelve IO ports of the eCOG1 are multiplexed among external peripherals to maintain a low pin count on the chip. They must be configured to accept external signals between the CPU and desired devices [EHM:88] (see Figure 3.5.). Timers, DUART, DUSART (supporting six serial protocols, two of which can be selected at a time), EHI, and EMI have dedicated channels to the port configurator. Other devices can be configured through General Purpose IO (GPIO) or Parallel IO (PIO). GPIO signals are controlled individually. PIO controls sixteen IO signals in parallel (sometimes called "a 16-bit parallel IO bus").

Figure 3.5. The Port Configurator [EHM:88]



Each IO port, denoted by a letter from A through L, contains either four or eight bits. A register for a port selects peripheral signals that are routed to the pins (*rg.port.sel1* or *rg.port.sel2*) and a whole port can be enabled or disabled (*rg.port.en* or *rg.port.dis*). More than one pin can be configured for the same peripheral signal. Complete configuration rules are presented in appendices D through G of [EHM]. For low power consumption unused pins are driven low rather than disabled.

GPIO

There are 29 GPIO signals from GPIO0 to GPIO28, each of which is individually configured as input or output. This is done via two set-clear pairs of bits (in *rg.io.gp..._out*) for output control: enable/disable and set/clear. Writing a one to the *fd.io.gp..._out.en...* bit enables the output; writing a one to the *fd.io.gp..._out.dis...* disables the output and enables the input. When the gpio is enabled, writing a one to *fd.io.gp..._out.set...* causes the gpio to be driven high; writing a one to *fd.io.gp..._out.clr...* causes the gpio to be driven low.

After enabling the output, the current value of the signal is set according to the previous write, or low if immediately after reset. As enabling and disabling (for writing/reading) a GPIO are directed to different bits, the virtual machine maps both registers as custom and monitors each write. When a GPIO generates an interrupt (for example, for Ethernet) the respective *fd.io.gp..._sts* register field will be set and then cleared. Besides the interrupt status, the status register (*rg.io.gp..._sts*) contains the current value at the pin.

Interrupts for each GPIO are configured in *fd.io.gp..._cfg.int...* to one of the following values:

"0 Disabled
1 Reserved
2 Low level
3 High level
4 Rising edge
5 Falling edge
6 Reserved
7 Rising or Falling edge" [EHM]

If the *fd.io.gp..._cfg.wake_en...* field contains a one, a wake up signal will be generated when the processor is asleep and the state of the associated GPIO signal changes. If the same field contains a zero, the processor is not awaken.

The GPIO registers can thus be summarized as follows:
rg.io.gp..._cfg - configures interrupts and processor wake-ups;
rg.io.gp..._out - output control;
rg.io.gp..._sts - status information for interrupts and the signal value.

Clock Devices

Introduction

A selection of clocks is generated by two oscillators and two PLLs (phase-locked loops). The oscillators, or reference clock sources, provide two clocks running at HIGH and LOW REF frequencies. They can optionally be input into the PLLs to obtain a further two clocks - the HIGH PLL and LOW PLL clocks.

The clocks are configured as the sources for dividers (for CPU and memory device clocks) or ripple counters/ divide chains (for peripheral clocks). There are (peripheral) clocks for timers, DUART, DUSART, and ADC. The flash, cache, EMI, and EHI are the memory devices that are configured by the

CPU/memory selector block. The dividers and divide chains multiplex the four clock sources into a greater variety of clock frequencies, which is necessary to accommodate devices of different speeds and provide power-saving capabilities [EHM:66,67] (see Figures 3.6. and 3.7.).

Figure 3.6. The System Support Module [EHM:66]

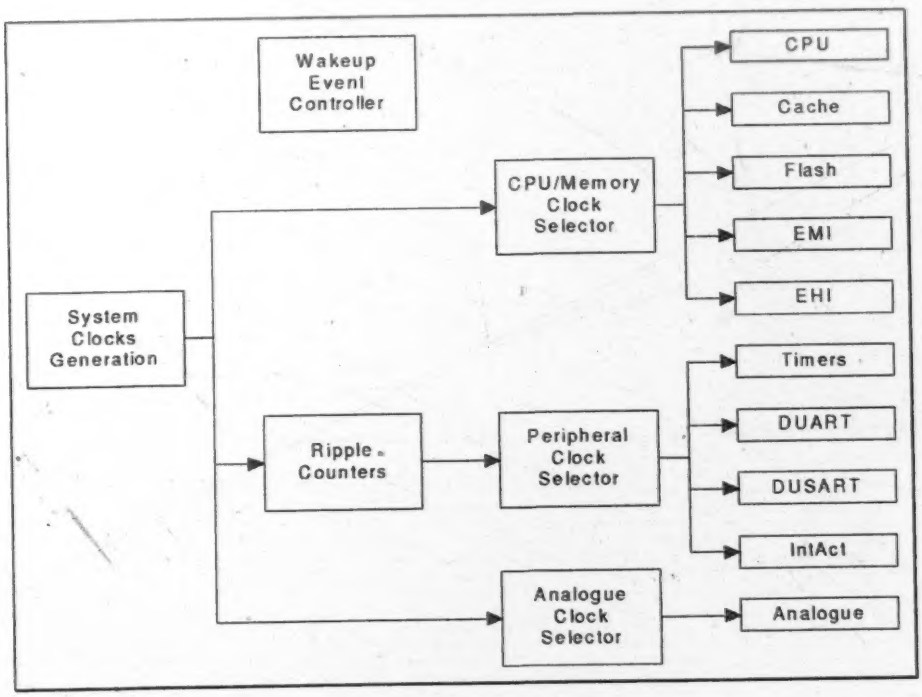
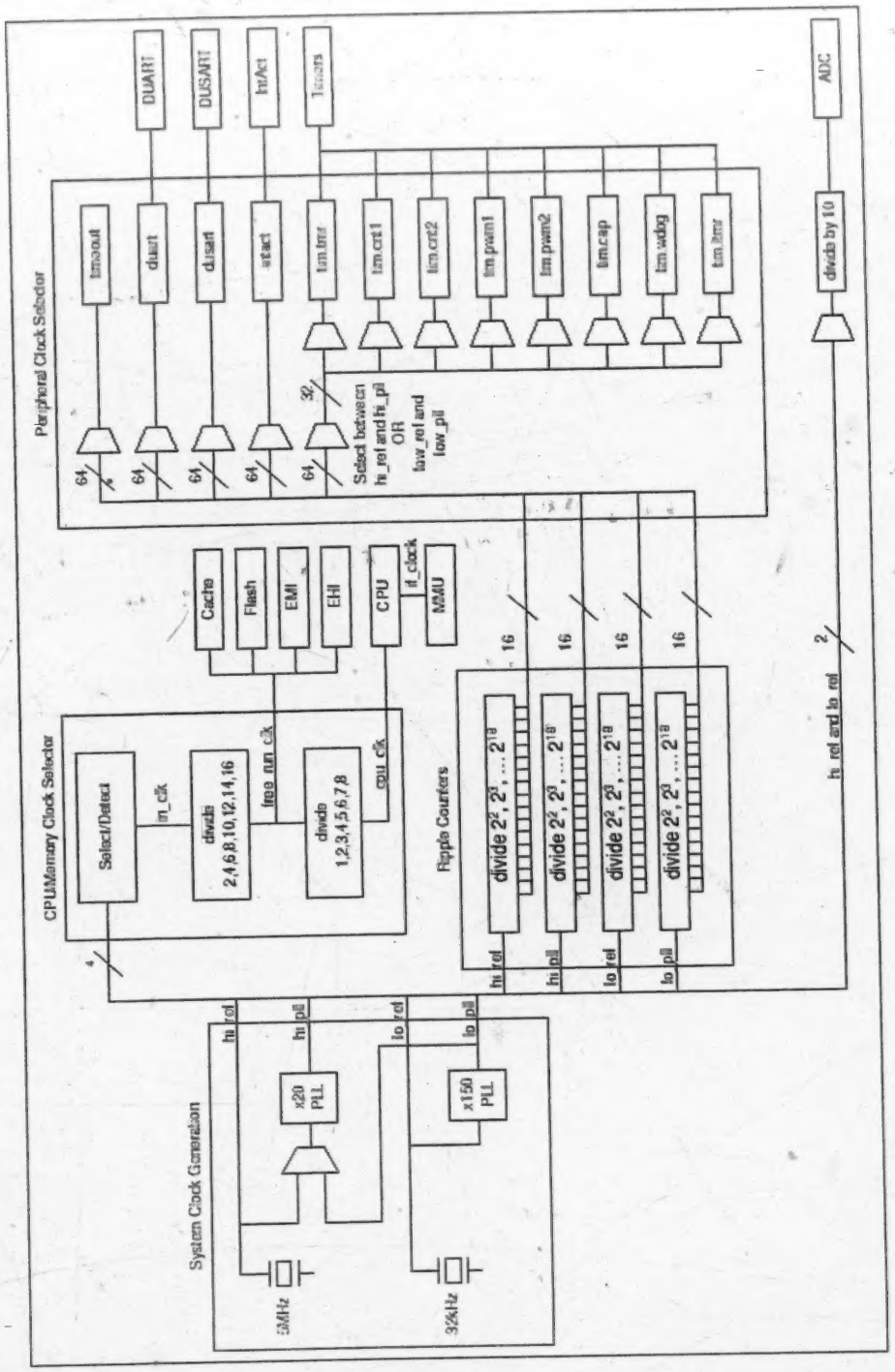


Figure 3.7. Clocking In Detail [EHM:67]



The separate clock domains are controlled by a set/clear register pair in the Reset system support module. Thus, the clocks are managed by the enable/disable and set/reset pairs which are standard for devices, except that the set/reset pair is in a separate Reset module and the usual semantics attached to the set and clear operations are reversed. In addition, the clocks are reset and cleared while disabled. In other words, reset is similar to the usual disable, clear is similar to enable; enable can be thought of as a set, and disable as a clear.

CPU Clock

To configure the CPU clock, reset is cleared and the respective (HIGH) clock and PLL are enabled. Further, the dividers are initialized to obtain a desired frequency for the CPU clock. The CPU clock frequency in hertz (Hz) is computed as a function of the CPU clock selector and divider values (in *rg.ssm.cpu*):

Figure 3.8. CPU Clock Frequency Computation [Author: E.Akhmetshina]

```
switch ((rg.ssm.cpu & SSM_CPU_STS_MASK) >> 8) {
  case SSM_CPU_STS_HIGH_REF_CLK:
   cpuclockfreq = HR;        //High Ref
   break;
  case SSM_CPU_STS_LOW_REF_CLK:
   cpuclockfreq = LR;        //Low Ref
   break;
  case SSM_CPU_STS_HIGH_PLL_CLK:
   cpuclockfreq = HP;        //High PLL
   break;
  case SSM_CPU_STS_LOW_PLL_CLK:
   core_read_mem (rg2addr (rg.ssm.cfg), &val);
   if (val & SSM_CFG_PLL_STEPUP_MASK)  {
    cpuclockfreq = PS;       //PLL Step-up
   } else {
    cpuclockfreq = LP;       //Low PLL
   }
```

```
    break;
  }//switch

  cpuclockfreq /= ( 16- 2* (rg.ssm.cpu & SSM_CPU_PRESCALAR_MASK) );
  cpuclockfreq /= ( 8- ((rg.ssm.cpu & SSM_CPU_CPU_CLK_DIV_MASK)
>>3) );

  return cpuclockfreq;
```

ecogsim keeps track of the elapsed processor clock cycles that are used as a measure of time. Therefore, interrupts for the clock devices are modeled based on the current value of processor cycles, the CPU clock frequency in Hz, and the respective device frequency in Hz. By taking the ratio of the CPU clock frequency in Hz and the respective device frequency in Hz we calculate in how many processor clock cycles interrupts should occur and compare that value with the current number of processor cycles.

Timer (RTC)

The frequency of timers, as that of the CPU clock, is configured in the System Support Module. The Timer block provides a further control over the interrupt frequency. For example, for the run time clock (TMR), an interrupt is generated when the *tmr* counter value reaches zero. This *tmr* counter value is programmed via the timer load register, and can be reloaded upon reaching zero back to the set value either automatically (implicitly) or manually (explicitly).

For the RTC to be initialized, the clock input source for it must be configured and enabled in the SSM. The timer enable, load, and command registers are set in the Timer block [EHM:143], and then timer interrupts are enabled.

Below is how the clock frequency in Hz can be determined for any timer device (such as the run time clock or a counter):

Figure 3.9. Timer Frequency Computation [Author: E.Akhmetshina]

```
core_read_mem (rg2addr (rg.ssm.div_sel), &val);
if (val & SSM_DIV_SEL_LOW_CLK_TIMER_MASK) {//low_clk
  if (val & ssm_div_sel_pll_mask) {//pll
    timerfreq = LP;
  } else { //ref
    timerfreq = LR;
  }
 } else { //high_clk
  if (val & ssm_div_sel_pll_mask) {//pll
    timerfreq = HP;
  } else { //ref
    timerfreq = HR;
  }
 }

 core_read_mem (ssm_tap_sel_rg_addr, &val);
 timerfreq >>= ( ((val & ssm_tap_sel_mask) >> 8) +2 ) ; //ripple
counters/divide chains

 core_read_mem (tim_ld_rg_addr, &val);
 timerfreq /= (val +1); //in how many tmr clock periods timer interrupt is fired

 return timerfreq;
```

The *rg.tim.int_en1* and *rg.tim.int_dis1* registers are mapped as custom to monitor the status of timer interrupts. The *rg.ssm.cpu* register is also custom mapped for updates to the CPU clock status based on values written to the *rg.ssm.cfg* register.

Ethernet

The Ethernet registers are memory-mapped by PicOS (using the EMI) starting at address 0x3d80. Of the eight words (0x3d80 to 0x3d87), 0x3d80 and 0x3d82 to 0x3d84 are custom-mapped. The others - 0x3d81 and 0x3d85 to 0x3d87 - are ram-mapped. The custom locations are needed to model the MAC address (the address register in Bank 1), data for reading and writing (the data

register in Bank 2), the receive status (the Rx FIFO register in Bank 2), and the MMU (the MMU command register in Bank 0).

The Ethernet chip has its registers multiplexed in four banks, Bank 0 to Bank 3. A bank is chosen using a special register select register. For the custom memory locations, the virtual machine keeps values at the locations for the four banks. For each memory access, it reads the register select register to determine the semantics of the operation and carries it out.

The MII (Media Independent Interface), or a MAC-PHY interface, is the mechanism used to interface (possibly external) PHY physical media to the MAC implementation in the chip. The physical registers (PHY, as opposed to MAC) are read and written through the MII register (in Bank 3) of the chip, and are not modeled in the emulation. In fact, those operations are configured out in the system to save on execution time.

The networking devices communicate in the emulation via shared memory utilities provided by the Windows operating system. Each destination node is assigned a data structure that contains a node index, node coordinates, a radio range, a radio FIFO, a receive FIFO of Ethernet frames, an Ethernet Receive Control Register (RCR), and a MAC address. Each of the three words of the MAC address of a node is set to its node index.

For writing to the Ethernet, the destination MAC address is determined from the frame contents. As each node contains its own Receive Control Register, when transmitting "through shared memory" we can select which nodes to transmit to. For example, if the destination MAC address is a multicast address (the one used in PicOS) then we transmit to all the nodes (except the source node) whose RCR is set to accept multicast frames. If the address is not a multicast then we Tx to the destination determined by the destination MAC address (if it is found in our pool of nodes with "legal" MAC addresses), as well as to the nodes whose RCR value tells us that the node is promiscuous.

Transmitting through shared memory involves dropping a frame in the receive FIFO. The MMU is modeled for transceiving in such a way that the chip is always ready to write/transmit.

For receiving from the Ethernet, an interrupt is generated when the Ethernet receive FIFO is not empty. Since both Ethernet interrupts and radio XEMICS interrupts are gpio interrupts, they share the same gpio irq. Thus if Ethernet interrupts are enabled (checked by reading the ram-mapped Ether interrupt register from the ecogsim core), the virtual machine "drives high" the respective gpio interrupt status pin and sets the gpio irq. The gpio interrupt is picked up by the PicOS gpio interrupt handler. Radio XEMICS interrupts are handled in a similar manner except that a different gpio is used.

DUART

The eCOG board has two DUARTs: DUART A and DUART B. They are meant to be analogous to each other (even though with some accidental inconsistencies for DUART B). However, in the absence of actual serial cables that connect devices, they are emulated differently. In the emulation, DUART A is assumed for connection with the terminal and DUART B is for connection with peer boards.

For user input, the virtual machine provides a custom command that is invoked from the ecogsim worksheet:

> ui "<*command directed to the board*>",

where <*command directed to the board*> is an application-specific string of input characters virtually transmitted over a serial connection to the eCOG from a PC. (Output from most eCOG devices, including DUART A, appears in the same *ecogsim* worksheet.)

In the emulation, this reception is always handled by DUART A. If DUART A interrupts are enabled (via the *rg.duart.a_int_en* register), the DUART A receive ready IRQ is virtually asserted and the PicOS DUART interrupt handler wakes up the DUART A driver for reading. If interrupts are disabled (via the *rg.duart.a_int_dis* register), the interrupt is missed, but the reading may happen the next time the DUART driver is executing, perhaps upon waking up after a regular delay. Note that each new invocation of the *ui* command overwrites the emulated DUART A buffer.

The emulation assumes that either DUART is always write-ready, i.e. the 16-bit (*rg.duart.[a,b]_tx16*) and 8-bit (*rg.duart.[a,b]_tx8*) transmit registers are emptied instantaneously. It is read-ready when data is available for reading, i.e. when the 16-bit receive register (*rg.duart.[a,b]_rx*) is not full. This is implemented when the custom-mapped *rg.duart.a_sts* or *rg.duart.b_sts* registers are read.

Whereas receive ready interrupts for DUART A are generated in the custom *ui* function after the user inputs a string of characters, DUART B receive ready interrupts are generated after a byte (*rg.duart.b_tx8*) or a word (*rg.duart.b_tx16*) of data has been written to the DUART. Each time a check is made that DUART B interrupts are enabled (*rg.duart.b_int_en*, *rg.duart.b_int_dis*). Following this design decision, DUART B virtually communicates with itself and not with DUART B of a different node. In other words, both a receiver and a transmitter processes must run on the same node.

The DUART transmission (baud) rate (*rg.duart.a_baud, rg.duart.b_baud*) is not modeled in the present version of the virtual machine because of the already slow system speed in the emulated environment. The frame parameters, or protocol specifics for serial transceiving at the physical signal level, are configured in the initialization function of the PicOS DUART driver. As they can not be changed dynamically, they are not emulated.

Radios

The custom "*node*" command allows the user to set or query the following parameters of a node:

> node *[<i> or <i> <x> <y> <r>]*,

where i is the node index, x is the x coordinate, y is the y coordinate, and r is the radio range (for transmission). (The "<" and ">" in the notation denotes "the value of". The "[" and ']" means "optional".) These node parameters are set to default values at the initialization of the DLL and displayed in the *ecogsim* window at that time. The x and y coordinates are initialized to equal the node index, and r to equal 2. The values can be easily changed, for example, for a communication protocol evaluation study, from the *ecogsim* worksheet or a batch file.

To get the current values of the node parameters, the user can just type:
> node
or
> node <*i*>, where *i* is a desired node index.

> node
will output parameter values for the current node (i.e. the node in whose *ecogsim* window the command is typed).

> node <*i*>
will output parameter values for node *i*.

For each supported radio board (XEMICS, RFMI, or VALERT), the virtual machine custom-maps a GPIO output register for transmitting and a GPIO status register for receiving. Given that the XEMICS board "overrides" the output register (*rg.io.gp0_3_out*) used by the eCOG LEDs, both devices can

not be configured in by an application. For an application requiring the LEDs, the radio type configuration option (*RADIO_TYPE* in *sysio.h*) must be set to anything other than *RADIO_XEMICS*. Possible options are *RADIO_RFMI*, *RADIO_VALERT*, or *RADIO_NONE*.

The unit of virtual radio transceiving "via shared memory" is one bit. This is different; for example, from Ethernet, for which the unit of emulated transceiving is one frame. The bit is transmitted to all nodes in the current range of the sender, based on the values of the x and y coordinates of the nodes, except the sender itself. If the radio is enabled for transmitting, a 1 or a 0, depending on whether a set or clear GPIO pin is asserted, is deposited in the radio "FIFO" of the receivers. The pin is then "cleared", i.e. "driven low".

The radios operate in half-duplex, i.e. they can either only transmit or receive at any given time. Therefore, the virtual machine maintains their enable status (RCV or XMT). For the RFMI or VALERT, the radio is enabled for receiving or transmitting by asserting a GPIO pin. The procedure for enabling the XEMICS is more involved, as stipulated by the more advanced chip [XEM]. It also includes setting up a timer (the counter *CNT1*) for software control of the transmission/baud rate. We use the counter status (via *rg.tim.ctrl_en* and *rg.tim.ctrl_dis*) as a proxy for enabling for transmitting.

The eCOG receives radio signals from the XEMICS as a bit stream that is already synchronized in the hardware. Whenever the bit synchronizer is turned on, software does not need to implement encoding protocols (except that the bit synchronizer needs at least one transition every 8 bits for a payload and one transition every bit for a 24-bit preamble). Its synchronized *DATAOUT* and *DCLK* values are available as *fd.io.gp8_15_sts.sts14* and *fd.io.gp8_15_sts.sts15* respectively. The value of *DATAOUT* can simply be sampled at the rising edge of *DCLK*. In the emulation, whenever *DCLK* is high (which happens when data is received) the virtual machine signals that *DATAOUT* is valid.

The other two radio chips, on the contrary, require a more elaborate software signal processing. Unlike for the XEMICS, the received radio signal is not synchronized. When the timer interrupt detects a high signal, a change in the signal value is monitored as well as its duration. A signal is considered to be received when it changes in value or when it has been at the same level for a maximum period of time. This period of time is calibrated for a given bit rate by using a software timer, as is the signal duration for transmitting. In the emulation, the (continuous) signal "history" is represented as a (discrete) sequence of 0's and 1's, where the duration of each signal is the same, or equal one. By way of example, three high signals followed by a low signal are represented as 1110.

Another notable feature of the XE transceiver is that it operates in one of four programmable operating modes: sleep, standby, receiver, and transmitter. Each of the modes is characterized by a different current consumption level, so that, in addition to switching between transmitting and receiving, the modes can be switched for power management. The MODE [2:0] pins select a mode when EN is low (*fd.io.gp20_23_out.set22*), and the change is applied when EN is high (*fd.io.gp20_23_out.clr22*). Three pins are used (as opposed to two) as some intermediate mode values are necessary for transition between the major modes. Switching into the receiver or transmitter mode from sleep or standby is done as prescribed by the transition sequences [XEM].

The XEMICS configuration and status (physical) registers can be used, for example, to set the bit rate or the clock frequency, to turn the bit synchronizer on, to configure the receive pattern recognition feature used to interrupt the microcontroller when a preamble is detected, etc. They are written and read by the eCOG via a three-wire serial bus (pins SI and SCK from the eCOG to the XE and pin SO from the XE to the eCOG) following a specific protocol [XEM:14]. This communication is not emulated in the DLL, but rather the values are kept in the respective variables in the ecogsim core.

Another caveat is in order. When interfacing the XE to the eCOG, MODE [2:0] are multiplexed with SCK, SI, and DATAIN for a minimum interface. Therefore they use the same gpio locations (gpio 0, gpio 11, and gpio 12 respectively). This overlapping however does not lead to a conflict in the emulation as the modes are changed when the transmitter is not enabled.

LCD

The LCD driver is notable for its numerous delays (i.e. because of its *delay()* function calls) in proceeding between its states. It was prototyped in the virtual machine to follow the suit of the complete real LCD driver at first (as still can be available for a debugging configuration). Later, to speed up the system in the emulated environment, higher levels of the driver had to be bypassed. At present, the only function of the driver that is invoked for the virtual machine from the PicOS library is the lowest-level function that does the actual writing on the LCD. (This is found in the library file *sim_lcd.c*.)

A character is written on the LCD by driving GPIO 17,18,19, and 20 for a nibble of the byte. This happens when data as opposed to a command is sent (GPIO 8 is set as opposed to cleared). Possible LCD commands are seek to location (1,1) (corresponds to the byte 0x02), seek one to the right (0x14), clear (0x01), etc.

LEDs

The LEDs are emulated by displaying the value written to the *rg.io.gp0_3_out* register. As the led system process in the PicOS library may periodically write to the register and all emulated devices output to the same ecogsim worksheet, just updates to the register are shown.

Beeper

The beeper tone is emulated by reading the *fd.ssm.tap_sel2.pwm1* value which corresponds to the divide chain for a Pulse Width Modulated timer. The tone duration is programmed by resetting the timer at the end of the duration. Only the tone is displayed because a message at the end of its duration would interfere with output from the other emulated devices.

This concludes the description of the virtual machine. The next section describes a system scheduling study and implementation undertaken using the virtual machine.

4. Process Scheduling Using the Virtual Machine

In this section we demonstrate how the virtual machine was used to add a new scheduling policy to the PicOS kernel. The prototyping and testing were done completely in the emulated environment, with no need for real eCOG hardware. On the contrary, if the loader/debugger were used for system development, the eCOG flash would need to be burned each time the code is changed and tested. For testing, we developed a simple application consisting of processes that deploy eCOG peripherals, as illustrated below.

The original kernel supported one scheduling policy. It was based on the order the processes were created, or the order process control blocks (PCB's) were assigned to them. To select a ready task for execution the scheduler traversed the list of PCB's starting from index 0. Thus, the closer the PCB was located to the start of the list, the higher its implied or implicit priority was. The system did not allow processes to change their own priority (or location in the PCB list) or the priority of another process.

The new configuration supports a priority-based scheduling where priorities can be explicitly assigned by the system or application. The choice of scheduling is controlled by an option (*SCHED_PRIO*) in a header file (*sysio.h*). If the option is not set, the system scheduling defaults to the original scheduling. If it is set, the scheduler will run a ready task of the highest priority.

Priorities are assigned as follows. When a process is created (by the *fork()* operation), its priority field is initialized to the adjusted index of the process in the PCB list. The index is adjusted by a factor of 10. In this way, an application is free to "insert" application level tasks anywhere in the priority list, even ahead of system tasks such as device drivers (this, however, is not recommended). Furthermore, process priorities need not be unique. Ready

processes with equal priorities are resolved in the standard PicOS way, i.e. a process whose PCB index (or, equally, process id) is least is scheduled first. Note that a higher priority corresponds to a lower value of the priority field. In other words, priority 0 is the highest, and priority equal the maximum number of tasks times 10 is the lowest.

The kernel now provides operations for manipulating process priorities, *priotitizeall ()* and *prioritize ()*. *prioritizeall ()* can be used to assign a new priority value to all instances of a PicOS process (identifiable by its code). *prioritize ()* can be used to assign a priority level to one instance of a PicOS process (identifiable by its process id). The functions have the following interface.

---> int prioritize (int pid, int pr)

This function assigns a scheduling priority to the indicated process. As a special case, if the process id is passed as 0 the current (invoking) process is assumed. Otherwise, if the process id is valid and the priority value is within the legitimate range (from 0 inclusive to the maximum number of tasks times 10 exclusive), the priority is updated for the process. Otherwise, if the passed priority value is not legitimate it is not set. In any case, the function returns the process priority. For example, a value of -1 can be passed as the priority argument to get the current priority value.

---> int prioritizeall (code_t fun, int pr)

This function assigns a scheduling priority to all process instances of the indicated code function and returns the number of processes whose priority value has been changed. If no processes of the indicated code function are found, the function returns 0. The priority value must be within the legitimate range (from 0 inclusive to the maximum number of tasks times 10 exclusive).

The functionality is illustrated (and tested) with the following example.

Figure 4.1. A Priority Test Application [Author: E. Akhmetshina]

```
#include "sysio.h"

heapmem {100};

#include "lcd.h"        //LCD functions header
#include "ser.h"        //Serial functions header

#define RS_INIT  00
#define RS_RCMD  10
#define RS_LCD  20
#define RS_PRIO  30

#define IBUFSIZE 128


process (A, void)

 nodata;

 entry (RS_INIT)
   upd_lcd ("1");      //update the LCD to display "1"

 entry (RS_INIT+1)
   finish;

endprocess (3)


process (B, void)

 nodata;

 entry (RS_INIT)
    upd_lcd ("2");     //update the LCD to display "2"

 entry (RS_INIT+1)
   finish;

endprocess (3)


process (root, void)
/* ========================================== */
```

```
/* This is the main program of the application */
/* ========================================== */

static char *ibuf = NULL;
static word p1, p2, pid;

nodata;

entry (RS_INIT)

      if (ibuf == NULL)
         ibuf = (char*) umalloc (IBUFSIZE); //dynamically allocate memory

entry (RS_RCMD-1)

   ser_out (RS_RCMD-1, "\r\n"
"         'p p1 p2 (priority test)\r\n" ); //serially output a string

entry (RS_RCMD)

   ser_in (RS_RCMD, ibuf, IBUFSIZE); //serially input a string

   if (ibuf [0] == 'p')
      proceed (RS_PRIO); //proceed to a state

entry (RS_LCD-1)

   ser_out (RS_LCD-1, "Illegal command or parameter\r\n"); //serially output
   proceed (RS_RCMD-1);

entry (RS_PRIO)

   if (scan (ibuf + 1, "%u %u", &p1, &p2) != 2)
      proceed (RS_LCD-1);

   diag ("Changing priority %d ", prioritize(0,-1));  //diagnostic serial output
   diag ("for %d root process(es) to ", prioritizeall(root,p1));
   diag ("%d", prioritize(0,-1));
   diag ("Changing priority to %d", prioritize(0,p2) );
   diag ("The new priority is %d", prioritize(0,-1));

entry (RS_PRIO+1)

   pid =fork (A, NULL);        //create a process and test its priority
```

```
  diag ("Changing priority %d ", prioritize(pid,-1));
  diag ("for %d A process(es) to ", prioritizeall(A,p1));
  diag ("%d", prioritize(pid,-1));
  diag ("Changing priority to %d", prioritize(pid,p1+1) );
  diag ("The new priority is %d", prioritize(pid,-1));

entry (RS_PRIO+2)

  pid =fork (B, NULL); //create a process and test its priority

  diag ("Changing priority %d ", prioritize(pid,-1));
  diag ("for %d B process(es) to ", prioritizeall(B,p2));
  diag ("%d", prioritize(pid,-1));
  diag ("Changing priority to %d", prioritize(pid,p2+1) );
  diag ("The new priority is %d", prioritize(pid,-1));

  proceed (RS_RCMD);

endprocess (1)
```

The application includes the standard *sysio.h* header file, as well as specific header files for the required IO devices, *lcd.h* and *ser.h*. The process entries or states are defined as constants with a convenient lag of 10 for a "cushion" to include additional states in between as the need arises.

There are three processes in this application. The *root* process is typical for an application, each must define one. It is similar to the *main()* function. Its second, data, argument is void so that it declares *nodata*. The body of this, like that of any other process, is a set of states. The states and transitions among them comprise a finite state machine. In the initial state, this root process dynamically allocates a memory buffer using *umalloc()*. It automatically, by default, transits to the next state, *RS_RCMD-1*, to serially output a message to the user through a currently chosen DUART (DUART A since the initialization). The first argument of the *ser_out* function call specifies a state (the same state *RS_RCMD-1* in this case) to proceed to after a delay, if the call can not successfully return immediately, i.e. if it blocks.

Once again, the *root* process transits through the next state to serially read user input into a locally declared buffer, *ibuf*, of *IBUFSIZE*. If the user inputs a legal command, the process *proceed()*'s to state *RS_PRIO*. As usual, the *proceed()* call does not take the process to the indicated state immediately, but rather through the scheduler, i.e. the next time this process is scheduled for execution. Thus, *proceed()* serves as a point the process is preempted voluntarily, or co-operatively with the other processes. If the user input was not legal, the process "slides" through state *RS_LCD-1* to output a warning message and return to state *RS_RCMD -1* to remind the user of the legal command format.

The states *RS_PRIO*, R*S_PRIO+1*, and *RS_PRIO*+2 serve in this application to run a series of *prioritize()* and *prioritizeall()* operations. The operations set the scheduling priorities to the values last time input by the user. In the *RS_PRIO* state the priority for the root process itself (the caller) is changed, while in the other two states the priorities of two child processes are changed. Note that *prioritize()* is called for the *root* with the first argument of 0 to indicate that the call is for the currently running process, while for the child processes it is called with their respective process id's returned by *fork()*.

The *scan()* library function reads formatted typed values in from the input buffer string and checks the number of read values. To obtain the current priority value for a process instance, the priority argument of *prioritize()* is set to -1, which falls outside of the legitimate priority range. *Prioritizeall ()* for the *root* code function returns the number of root instances, rightfully 1 in this case. The *root* process is declared to have at most one instance at any time, as specified by the argument of *endprocess (1)*.

This is different for the two child processes, process *A* and process *B*, *fork()*'ed and manipulated in the root process, states *RS_PRIO+1* and *RS_PRIO+2* respectively. Both the process *A* and process *B* code functions are declared to run in at most three instances at a time. Processes *A* and *B* are very simple

processes that just write a "1" or "2" respectively on the LCD and explicitly end (the *finish* operation).

As they are *fork()*'ed in the *root*, their priorities default to the adjusted PCB index values. To demonstrate and test the new scheduling policy, we update the priorities to the ones input by the user in the *prioritize()* and *prioritizeall()* calls. The first input parameter refers to the priority of process *A*, and the second parameter to process *B*. When the application is run, the messages on the LCD appear in the order dictated by the input priority values. A process with a lower priority index is executed first.

5. Conclusions and Future Work

Emulation is a technique for deploying functionality from a non-native platform. In this thesis we presented an emulation system for PicOS, a multitasking embedded operating system designed to run on a microcontroller and drive peripheral devices. Together with a given emulator of the processor core, the IO system has been used to develop applications and operating system services in a virtual environment.

In SMURPH, a System for Modeling Unslotted Real-time Phenomena, a methodology was established for representing physical systems as a software event-driven specification. The primary goal of the system was to present a framework for simulating MAC-level networking devices and a platform for studying related communication protocols. The methodology implemented in the SMURPH kernel however was a contribution far too general to be confined to the limits of a network simulating package. It has inspired the next generation of systems. One of them is PicOS, a multitasking embedded operating system for reactive applications.

We examined the place of the paradigm in the state of the art in embedded operating systems. Further, we demonstrated a mechanism to represent the physical environment as emulated devices in a microcontroller domain that are managed by the PicOS kernel and operated by a reactive application. The range of devices is extended to include a variety of networking devices, vital devices such as the clock and memory, and basic devices such as the display.

As a result of this work, embedded network applications and protocols can be developed on the emulated platform. The emulated system approximates the application behavior that would be observed on the real hardware in the context of the PicOS operating system. This is achieved by utilizing a lowest-

level software interface to the hardware. The emulated IO system is considered as an instrument that may be useful for an easy adaptation of the PicOS platform and the novel programming paradigm that orchestrates it.

To demonstrate that the PicOS design and programming model are general and descriptive to accommodate a wide variety of platforms and applications, we are considering a different microcontroller environment. We plan to use a popular family of microprocessors such as Microchip's PIC18Fxxx (Peripheral Interface Controller). The PIC18Fxxx is chosen in consideration of PicOS' data and code requirements. Specifically, memory-wise this family of PICs may be closest to the eCOG1.

The PicOS operating system, including the interrupt service routines and device drivers, as well as the IO libraries are all written in C. They can be easily ported to a new architecture supported by an ANSI-compatible C compiler.

The PicOS was restructured for porting to include a separate architecture-independent directory (*kernel*). It consists of a kernel C file (*kernel.c*) and a kernel header file (*kernel.h*). The *Makefile*s were updated accordingly. The code that is eCOG-specific is put in the eCOG directory. Since the eCOG board is the only one envisioned for the eCOG1 at the moment, no separation between processor- and board-dependent parts is made. This separation is however a promising opportunity for the PICs that are represented by a great number of parts and boards.

At the moment of writing, the kernel is being compiled with a PIC compiler that claims to be ANSI C compatible (a quality ANSI-compatible compiler for the PICs has been viewed as a rarity). The compiler in addition offers a library of IO device driver routines for sample designs. All this coupled with the elegant PicOS design should facilitate a comfortable adaptation of the novel embedded programming paradigm by a wide community.

Bibliography

[Z1] Indiresan A., Mehra A., Shin K. "END: A Network Adapter Design Tool". IEEE INFOCOM, 1997.

[Z2] Indiresan A., Mehra A., Shin K. "The END: Exploring QoS Issues in Adapter Design via an Emulated Network Device". Submitted for publication, 1996.

[Z3] Rosenblum M., Bugnion E., Devine S., Herrod S. "Using the SimOS Machine Simulator to Study Complex Computer Systems". ACM Transactions on Modeling and Computer Simulation, vol. 7, no. 1, January 1997, pp.78-103.

[Z4] Dobosiewicz W. and Gburzynski P. "Protocol Design in SMURPH." In State-of-the-art in Performance Modeling and Simulation Vol. 1: Computer and Communication Networks, J. Walrand and K. Bagchi editors, Gordon and Breach, 1997, pp. 255-274.

[Z5] Akhmetshina E., Gburzynski P., and Vizeacoumar F. "PicOS: A Tiny Operating System for Extremely Small Embedded Platforms." Proceedings of ESA'03, Las Vegas, June 23-26, pp. 116-122.

[Z6] Gburzynski P., Maitan J., Hillyer L. "Virtual Prototyping of Reactive Systems in SIDE." Proceedings of the 5th European Concurrent Engineering Conference ECEC'98, Erlangen-Nuremberg, Germany, April 26-29, 1998, pp. 75-79.

[Z7] Gburzynski P. "Protocol Design for Local and Metropolitan Area Networks." Prentice Hall, 1996.

[Z8] Rosenblum M., Herrod S., Witchel E., Gupta A. "Complete Computer System Simulation: The SimOS Approach". IEEE Parallel and Distributed Technology. Accepted for publication.

[Z9] Won C., Lee B., Yu C. "Linux/SimOS A Simulation Environment for Evaluating High-Speed Communication Systems". ICPP. 2002

[Z10] Bedichek R. "Some Efficient Architecture Simulation Techniques". Proceedings of the USENIX Winter 1990 Technical Conference. 1990.

[Z11] Gburzynski P. and Maitan J. “Specifying Control Programs for Reactive Systems”. Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA'98, Las Vegas, July 13-16, 1998, pp. 1702-1709.

[Z12] Gburzynski P. “An Overview of SMURPH: an Object-Oriented Configurable Simulator for Low-level Communication Protocols”. 1997.

[U1] Labrosse J. "uCOS: The Real-Time Kernel". R&D Publications, 1992.
[U2] Labrosse J. "A Portable Real-Time Kernel in C". Embedded Systems Programming, v.5:no.5, May 1992, pp.40-53.
[U3] Labrosse J. "Implementing a real-Time Kernel". Embedded Systems Programming, v.5:no.6, June 1992, pp.44-49.

[ECM] Cyan Technology. "C Compiler User Manual". V1.5. 15 May 2002.
[EHM] Cyan Technology. "eCOG1/eCOG1i 16-bit Processor User Manual". V 2.0. 4 October 2002.
[XEM] XEMICS. "Data sheet XE1202". 2003.

[DIC] Howe D. The Free On-line Dictionary of Computing. 1993-2003.